

# I

---

## Wordplay

---

### Sengen: A Sentence Generator

SENGEN makes up English sentences similar to the following ones:

PECULIAR BIRDS HATE JUMPING DOGS  
FAT WORMS HATE PECULIAR WORMS  
RED GUINEA PIGS TRIP FUZZY WUZZY DONKEYS  
FAT GEESE BITE JUMPING CATS

One of the questions you might ask is this: Does SENGEN make up sentences the way we do or the way we did when we first learned to talk or write? Another question you might ask is: What relationship does SENGEN have to understanding grammar? The first question is open to research and speculation. The second might be an easier one to answer. Often when I first discuss this project with children, they do not relate the programming process to the learning of grammar. Later as they use their programs, the children frequently exclaim: "So this is why they call words nouns and verbs!" They also begin to appreciate formal systems. Studying grammar by generating sentences that obey certain rules requires the programmer to become aware of rules as well as of their exceptions.

Since this program seems to make sensible sentences without knowing very much about grammar, children often develop an appreciation for cleverness. For example, SENGEN doesn't know that some words are singular and some are plural or that singular subjects should be matched with singular verbs; it does not know about verb tenses or pronominal relations. Its apparent intelligence comes from the programmer's choice of words and categories.

In the following examples, the nouns and verbs are all plurals and the verbs are all in the present tense.

SENGEN builds sentences from vocabulary lists of nouns, verbs, adjectives, connectives, and so on. It then assembles its selections according to some rule of grammar.

#### *Making the Program*

One strategy in making a program might be to concentrate on developing a random sentence generator that outputs only a verb. For example:

Go.  
Run.

---

By Cynthia Solomon.

**WORDPLAY**

To do this a procedure is needed to blindly (randomly) pick out a selection from a list of possibilities.

Let's make up a list of verbs and then make a procedure to select a word from the list. In this example, the procedure VERBS outputs the vocabulary list.

```
TO VERBS
OP [EAT SCARE LOVE HATE [LAUGH AT] TRIP BITE]
END
```

Whenever VERBS is called, it outputs that list.

```
PR VERBS
EAT SCARE LOVE HATE [LAUGH AT] TRIP BITE
```

```
PR FIRST VERBS
EAT
```

```
PR LAST BL VERBS
TRIP
```

What we now want is a procedure that will randomly choose one of the items in this list. Here is the plan for this task: use a number obtained from RANDOM to point to an item in the given list of choices. Then get that item from the list. PICK does this and outputs the selection.

```
TO PICK :LIST
OP SELECT RANDOM COUNT :LIST :LIST
END
```

PICK's input is a vocabulary list. PICK calls SELECT, giving it the list and a number indicating which item in the list SELECT is to output.

There is a slight problem. RANDOM outputs a number from 0 up to but not including its input number. Thus its output in PICK is always one less than the length of the list. We can fix that by adding 1 to RANDOM's output.

```
TO PICK :LIST
OP SELECT 1 + RANDOM COUNT :LIST :LIST
END
```

SELECT carries out its job recursively. When its input number is one, it outputs the first item from its input list. Otherwise, SELECT subtracts one from its input number and takes away the first item from its input list and continues the process until the item is found.

Here is what SELECT looks like.

```
TO SELECT :ITEM :LIST
IF :ITEM = 1 [OP FIRST :LIST]
OP SELECT :ITEM - 1 BF :LIST
END
```

Now we can try PICK.

```
PR PICK VERBS
SCARE
```

```
PR PICK VERBS
EAT
```



We could try it on different lists:

PR PICK [1 2 3 4]

3

PR PICK [A B C D]

A

PICK seems to work.

Let's make a procedure that outputs just a verb.

TO VERB

OP PICK VERBS

END

PR VERB

BITE

Now we can move on to building a sentence by first making a one-word sentence.

TO SEN

PR VERB

SEN

END

SEN

LAUGH AT

SCARE

LAUGH AT

EAT

and so on.

Our attempt at making a one-word sentence fails because of the verbs in the verb list. Only EAT can be used without an object. So if we want to make grammatical one-word sentences, we have to restrict our choice of verbs.

Now let's make a sentence with a subject and an object. Let's follow the pattern already set up for verbs and make two operations NOUNS and NOUN. NOUNS outputs a list of nouns.

TO NOUNS

OP [BOYS [DOGS AND CATS] PUPPIES [SIAMESE FIGHTING FISH]

GEESE BIRDS GIRLS [GUINEA PIGS][MICE AND GERBILS] WORMS

TEACHERS DONKEYS CLOWNS [BASEBALL PLAYERS]]

END

NOUN outputs one of the items from NOUNS.

TO NOUN

OP PICK NOUNS

END

PR NOUN

CLOWNS

**WORDPLAY**

All we need to do to make a sentence is the following:

```
PR (SE NOUN VERB NOUN)
SIAMESE FIGHTING FISH SCARE BOYS
```

Imagine we had miscategorized the vocabulary and NOUNS could output a list like

```
RED LAUGHING TORTOISE BOY
```

We might then get sentences like

```
RED SCARE LAUGHING
BOY EAT TORTOISE
```

This kind of bug is typical of the kind people run into when they first do this project. Usually, when people confront their bugs, they begin to appreciate rules of grammar and the fantastic power we derive from categorizing words.

We can now make a procedure that outputs a sentence.

```
TO SEN
OP (SE NOUN VERB NOUN)
END
```

```
PR SEN
BASEBALL PLAYERS EAT DONKEYS
```

SENGEN can print this output and continue the process.

```
TO SENGEN
PR SEN
PR []
SENGEN
END
```

***Extensions***

One extension is to add adjectives to the sentences.

```
TO ADJECTIVES
OP [RED FAT [FUZZY WUZZY] PECULIAR JUMPING]
END
```

```
TO ADJECTIVE
OP PICK ADJECTIVES
END
```

Edit SEN.

```
TO SEN
PR (SE ADJECTIVE NOUN VERB NOUN)
END
```



The sentences are getting more complicated, so it is time to introduce additional categories like NOUNPHRASE and VERBPHRASE. For example:

```
TO NOUNPHRASE
OP (SE ADJECTIVE NOUN)
END
```

```
TO VERBPHRASE
OP (SE VERB NOUN)
END
```

```
TO SEN
OP (SE NOUNPHRASE VERBPHRASE)
END
```

Another possibility is to link two simple sentences by using connectives:

```
TO CONNECTS
OP [BUT AND [EVEN THOUGH]]
END
```

```
TO CONNECT
OP PICK CONNECTS
END
```

Finally, you change SENGEN to include the new sentence:

```
TO SENGEN
PR (SE SEN CONNECT SEN)
PR []
SENGEN
END
```

#### SENGEN

```
FAT DOGS HATE DONKEYS EVEN THOUGH
    JUMPING BASEBALL PLAYERS SCARE BOYS
```

```
RED CATS EAT WORMS EVEN THOUGH
    FUNNY BUNNY BASEBALL PLAYERS LOVE BOYS
```

```
FAT MICE AND GERBILS SCARE TEACHERS
    AND FAT CLOWNS BITE MICE AND GERBILS
```

---

#### PROGRAM LISTING

---

```
TO SENGEN
PR (SE SEN CONNECT SEN)
PR []
SENGEN
END

TO SEN
OP (SE NOUNPHRASE VERBPHRASE)
END
```

```
TO CONNECTS
OP [BUT AND [EVEN THOUGH]]
END

TO CONNECT
OP PICK CONNECTS
END

TO NOUNPHRASE
OP (SE ADJECTIVE NOUN)
END
```

```
TO NOUN
OP PICK NOUNS
END
```

```
TO NOUNS
OP [BOYS [DOGS AND CATS] PUPPIES ►
   [SIAMESE FIGHTING FISH] GEESE ►
   BIRDS GIRLS [GUINEA PIGS][MICE ►
   AND GERBILS] WORMS TEACHERS ►
   DONKEYS CLOWNS [BASEBALL ►
   PLAYERS]]
END
```

```
TO VERB
OP PICK VERBS
END
```

```
TO VERBS
OP [EAT SCARE LOVE HATE [LAUGH AT] ►
   TRIP BITE]
END
```

```
TO ADJECTIVE
OP PICK ADJECTIVES
END
```

```
TO ADJECTIVES
OP [RED FAT [FUZZY WUZZY] PECULIAR ►
   JUMPING]
END
```

```
TO VERBPHRASE
OP (SE VERB NOUN)
END
```

```
TO PICK :LIST
OP SELECT 1 + RANDOM COUNT :LIST :LIST
END
```

```
TO SELECT :ITEM :LIST
IF :ITEM = 1 [OP FIRST :LIST]
OP SELECT :ITEM - 1 BF :LIST
END
```

## Argue

ARGUE carries on a dialogue with you. When you run ARGUE, it expects you to type a statement in the form I LOVE LEMONS or I HATE DOGS. ARGUE comes back with contrary statements. For example, if you make the statement I HATE DOGS, the program types

```
I LOVE DOGS
I HATE CATS
```

If it doesn't already know the opposite of a word, it asks you. For example, if you type I LOVE LEMONS and ARGUE does not know the opposite of LEMONS, it types

```
I HATE LEMONS
WHAT IS THE OPPOSITE OF LEMONS?
```

If you tell it ORANGES, it will type

```
I LOVE ORANGES
```

Here is a sample dialogue.



## ARGUE

--&gt;I LOVE SALT

I HATE SALT

I LOVE PEPPER

--&gt;I HATE CATS

I LOVE CATS

I HATE DOGS

--&gt;I HATE DOGS

I LOVE DOGS

I HATE CATS

--&gt;I LOVE LEMONS

I HATE LEMONS

WHAT IS THE OPPOSITE OF LEMONS?ORANGES

I LOVE ORANGES

--&gt;

ARGUE *Can Reply to Your Statements*

When you run ARGUE, it types an arrow to let you know that it is ready for you to type your statement, then calls ARGUEWITH. ARGUEWITH is given the statement you type as its input. ARGUE is recursive so this process continues.

```
TO ARGUE
TYPE [\-\-\>]
ARGUEWITH RL
ARGUE
END
```

ARGUEWITH prints two responses to your statement. First, it turns around your statement; if you say that you *love* something, ARGUEWITH says that it *hates* it, and if you say you *hate* something, ARGUEWITH says that it *loves* it. Second, it makes a statement about the opposite of the object you mentioned.

```
TO ARGUEWITH :STATEMENT
PRINT ( SE "I LOVE.HATE SECOND :STATEMENT LAST :STATEMENT )
PRINT ( SE "I SECOND :STATEMENT OPPOSITE LAST :STATEMENT )
END
```

The procedure LOVE.HATE sees whether its input is "LOVE or "HATE and outputs the other one.

```
TO LOVE.HATE :WORD
IF :WORD = "LOVE [OP "HATE]
IF :WORD = "HATE [OP "LOVE]
END
```

## WORDPLAY

The ARGUEWITH procedure works only with statements in the form I LOVE *something* or I HATE *something* because it assumes that the second word in your statement is LOVE or HATE and that the last word in your statement is something whose opposite it can find.

ARGUEWITH uses SECOND to grab the second word in a sentence.

```
TO SECOND :LIST
OP FIRST BF :LIST
END
```

The OPPOSITE procedure is the real guts of the ARGUE program. It takes a word as its input and outputs the opposite of that word.

### *The Program Keeps Track of Opposites*

How does the program know that *pepper* is the opposite of *salt*? Somehow, the ARGUE program has to have this information stored. We use variables to hold this information. For example, :SALT is PEPPER, :CATS is DOGS. This is how we have chosen to store the facts the program “knows.” We call this a *data base*. You can look at the data base for the ARGUE program by looking at all the variables in the workspace. Try:

```
PONS
MAKE "PEPPER "SALT
MAKE "SALT "PEPPER
MAKE "DOGS "CATS
MAKE "CATS "DOGS
MAKE "LIFE "MARRIAGE
MAKE "MARRIAGE "LIFE
MAKE "DARK "LIGHT
MAKE "LIGHT "DARK
```

These variables are loaded into the workspace with the ARGUE program.\*

To find out the opposite of something, for example DARK, we can say

```
PR :DARK
LIGHT
```

or

```
PR THING "DARK
LIGHT
```

What if we want to find out the opposite of LIGHT? There is no easy way to find out it is DARK unless we have another variable named LIGHT, with value DARK. So we can say

```
PR THING "LIGHT
DARK
```

We have set up a convention in our data base that we always put in both parts of a pair. That way, we don't end up in the funny situation where it

\*If you type in the procedures and there are no variables in the workspace, ARGUE will create these variables when it asks you for the opposites of things.



is easy to find out that the opposite of ROUGH is SMOOTH, but impossible to find out what the opposite of SMOOTH is. Our mental concept of opposite is that it "goes both ways," so we make our data base reflect that.

### *How the OPPOSITE Procedure Works*

With this kind of data base we can write a procedure to output the opposite of something. Here is a possible first version of the OPPOSITE procedure:

```
TO OPPOSITE :OBJECT
OP THING :OBJECT
END
```

This is a good example of needing to use THING rather than dots(.). The word of which OPPOSITE is trying to find the value is whatever :OBJECT is. For example, if :OBJECT is the word SALT, then the program is trying to find :SALT. It must do this indirectly by using THING :OBJECT.

This first version of OPPOSITE has a problem. It only works for words that are already in the data base. If you make a statement like I LOVE SUNSETS and there is no variable named SUNSETS, then this OPPOSITE procedure will get an error. To solve this problem, we use NAMEP to check for the existence of a variable named by :OBJECT. In this example :OBJECT is the word SUNSETS; the program checks whether there is already a variable named SUNSETS. If there isn't, you'd like the program to learn the opposite of SUNSETS and put it in the data base. Then it can go ahead and argue with you about sunsets. The procedure LEARNOPP does this. OPPOSITE calls LEARNOPP when it needs to.

```
TO OPPOSITE :OBJECT
IF NAMEP :OBJECT [OP THING :OBJECT]
PRINT ( SE [WHAT IS THE OPPOSITE OF] :OBJECT "? )
LEARNOPP :OBJECT FIRST RL
OP THING :OBJECT
END
```

```
TO LEARNOPP :OBJECT :OPP
MAKE :OBJECT :OPP
MAKE :OPP :OBJECT
END
```

When OPPOSITE tries to find the opposite of a word that is not in the data base, it asks the user for the opposite. After the user types the opposite, OPPOSITE passes both the problem word and its opposite to LEARNOPP. LEARNOPP puts that pair of words in the data base.

### *Now ARGUE Can Argue Pretty Well*

So ARGUE can keep going as it adds new words to its data base.

## WORDPLAY

## ARGUE

```
-->I HATE PEPPER
I LOVE PEPPER
I HATE SALT
-->I LOVE SUNSETS
I HATE SUNSETS
WHAT IS THE OPPOSITE OF SUNSETS?SUNRISES
I LOVE SUNRISES
-->I LOVE SUNRISES
I HATE SUNRISES
I LOVE SUNSETS
```

and so on.

If we look at the data base after this, we can see what has been added.

## PONS

```
MAKE "SUNRISES "SUNSETS
MAKE "SUNSETS "SUNRISES
MAKE "PEPPER "SALT
```

and so on.

In order for the program to "remember" this data base, these variables must be saved by SAVEing this workspace on a diskette.

## SUGGESTIONS

The ARGUE program assumes that the sentences you type in are going to be exactly in the form

```
I LOVE something
```

or

```
I HATE something
```

If they are not, an error occurs and the program stops. You could improve the program so that it checks for the right kinds of sentences and asks you to retype them if there are problems.

Maybe it could know about more emotion words such as DESIRE, LIKE, DISLIKE, DESPISE, DETEST.

If you try:

```
I LOVE GREEN PEAS
```

the program will say:

```
I HATE PEAS
```

and ask you for the opposite of PEAS. It will ignore the GREEN. You might make a better arguing program that tries to figure out if there is an adjective and finds its opposite, so it would do something sensible like

## ARGUE

```
-->I LOVE GREEN PEAS
I HATE GREEN PEAS
I LOVE RED PEAS
```



ARGUE doesn't have any mechanism for dealing with single objects described by more than one word, like ICE CREAM. Perhaps a special way to type these in might be added.

You might want to look at the Madlibs and Sengen projects for more ideas that have to do with taking apart and putting together sentences. You might want to look at the Animal Game project for an example of a program with a different kind of data base that also appears to learn some simple things.

---

### PROGRAM LISTING

---

TO ARGUE	TO OPPOSITE :OBJECT
TYPE [\-\-\>]	IF NAMEP :OBJECT [OP THING :OBJECT]
ARGUEWITH RL	PRINT ( SE [WHAT IS THE OPPOSITE OF] ►
ARGUE	:OBJECT "? )
END	LEARNOPP :OBJECT FIRST RL
	OP THING :OBJECT
	END
TO ARGUEWITH :STATEMENT	TO LEARNOPP :OBJECT :OPP
PRINT ( SE "I LOVE.HATE SECOND ►	MAKE :OBJECT :OPP
:STATEMENT LAST :STATEMENT )	MAKE :OPP :OBJECT
PRINT ( SE "I SECOND :STATEMENT ►	END
OPPOSITE LAST :STATEMENT )	
END	MAKE "PEPPER "SALT
	MAKE "SALT "PEPPER
TO LOVE.HATE :WORD	MAKE "DOGS "CATS
IF :WORD = "LOVE [OP "HATE]	MAKE "CATS "DOGS
IF :WORD = "HATE [OP "LOVE]	MAKE "LIFE "MARRIAGE
END	MAKE "MARRIAGE "LIFE
	MAKE "DARK "LIGHT
TO SECOND :LIST	MAKE "LIGHT "DARK
OP FIRST BF :LIST	MAKE "SUNRISES "SUNSETS
END	MAKE "SUNSETS "SUNRISES

---

## Animal Game

The animal game is a little like twenty questions: you think of an animal, and the game tries to guess it by asking yes-or-no questions.\*

What makes the game interesting is that it learns new animals. When it can't guess your animal, it asks you to teach it the animal and its distinguishing characteristic. By learning new questions and new animals, the game gets "smarter."

\*This animal game is a popular computer game. It first appeared about ten years ago. Since then many people have implemented it in various computer languages. This Logo program was inspired by Bernard Greenberg's unpublished LISP textbook.

## WORDPLAY

Here's a sample dialogue between the computer and a person playing the animal game. Everything the user types is boldface.

?ANIMALGAME

PICK AN ANIMAL, ANY ANIMAL

IS IT FURRY?

The player's secret animal is "dog."

YES

HERE'S MY GUESS: IS IT A CAT?

NO

I GIVE UP. WHAT IS IT?

A DOG

PLEASE TYPE IN A QUESTION

WHOSE ANSWER IS 'YES' FOR A DOG

AND 'NO' FOR A CAT

Here's where the game gets smarter.

DOES IT BARK?

DO YOU WANT TO PLAY AGAIN?

YES

PICK AN ANIMAL, ANY ANIMAL

IS IT FURRY?

The player's secret animal is "dog" again.

MAYBE

PLEASE ANSWER YES OR NO

IS IT FURRY?

YES

DOES IT BARK?

YES

HERE'S MY GUESS: IS IT A DOG?

YES

I WIN!

I WIN!

DO YOU WANT TO PLAY AGAIN?

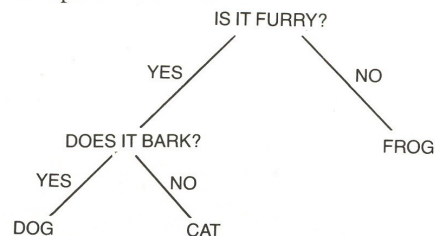
NO

?

Here's where the game asks the question it just learned!

### *Knowledge Grows on Trees*

Below is a diagram of the knowledge the game might have after someone has played it a few times. We call the diagram a *tree*, because it looks something like an upside-down tree.



The tree is made of questions and animal names. Each question has a "yes branch" and a "no branch." Each branch either leads to a question or ends at an animal name.

By drawing what the game knows in the form of a tree, we can get a more vivid picture of how the game works. For example, we can think of the game as exploring the tree from its top. It always starts at the IS



IS IT FURRY? question. Its goal is to climb down the branches to an animal name. The animal it finally reaches is the one it guesses.

Let's play an imaginary game and trace the game's progress on the tree. Our secret animal is "mouse."

The game's first question is always the question at the tree's top: IS IT FURRY? Since a mouse is furry, we answer yes.

The game follows IS IT FURRY?'s yes branch to the DOES IT BARK? question. From DOES IT BARK?, the game can descend to either of the furry animals, DOG or CAT, but it can no longer reach the unfurry animal, FROG. By descending IS IT FURRY?'s yes branch, the game has narrowed down its possible guesses to furry animals.

The game now asks the question DOES IT BARK?. A mouse does not bark, so we answer no.

The game follows DOES IT BARK?'s no branch to the animal name CAT. When the game reaches an animal name, it guesses that animal. Here, of course, the game's guess is wrong. To improve its chances of guessing right the next time, the game learns the player's secret animal. Before we look at the learning process, let's examine how the game represents its knowledge as lists.

### Making Trees with Logo Lists

Consider the very simple tree below. Here we represent it as a list.

```
[[IS IT FURRY?] CAT FROG]
```

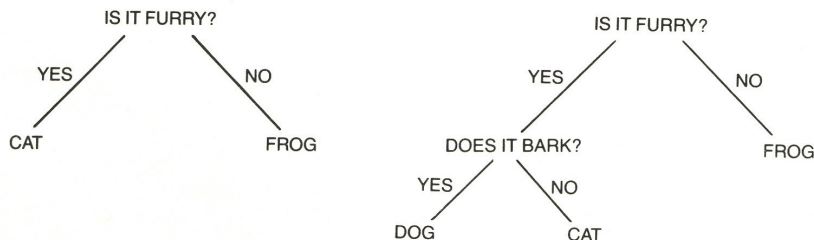
The tree is a list of three elements: a question, the question's yes branch, and the question's no branch. In this case, the question is [IS IT FURRY?], its yes branch is CAT, and its no branch is FROG.

Both branches of the left tree below are animal names. Sometimes, as we've seen, a branch does not lead directly to an animal name but to another question that has its own two branches; it leads, that is, to another tree or *subtree*.

For example, look now at the slightly more complicated tree. Here it is represented as a list.

```
[[IS IT FURRY?] [[DOES IT BARK?] DOG CAT] FROG]
```

This slightly more complicated tree is also a list of three elements: a question, its yes branch, and its no branch. The question is [IS IT FURRY?]; its yes branch is the subtree [[DOES IT BARK?] CAT DOG]; its no branch is the animal name FROG.



## WORDPLAY

## Examining Trees

We can write procedures that look at each of a tree's three parts. Sometimes we want to look at a subtree of a tree. Since a subtree is itself a tree, these procedures work on subtrees too. The procedures all expect a list of three elements as input.

```
TO QUESTION :TREE
OP FIRST :TREE
END
```

```
TO YES.BRANCH :TREE
OP FIRST BF :TREE
END
```

```
TO NO.BRANCH :TREE
OP FIRST BF BF :TREE
END
```

Here's an example of how they work.

```
?MAKE 'SAMPLE [[IS IT FURRY?] [[DOES
  IT BARK?] DOG CAT] FROG]
?SHOW QUESTION :SAMPLE
[IS IT FURRY?]
?SHOW YES.BRANCH :SAMPLE
[[DOES IT BARK?] DOG CAT]
?SHOW NO.BRANCH :SAMPLE
FROG
?SHOW NO.BRANCH YES.BRANCH :SAMPLE
CAT
?PR COUNT :SAMPLE
3
?PR COUNT YES.BRANCH :SAMPLE
3
```

## Exploring the Game's Knowledge

The animal game's first task is to begin at the tree's top and follow branches to a guess. The procedure that does this is called EXPLORE.

```
TO EXPLORE :TREE
IF WORDP :TREE [FINISH.UP :TREE STOP]
IF YESP QUESTION :TREE
  [EXPLORE YES.BRANCH :TREE]
  [EXPLORE NO.BRANCH :TREE]
END
```

The first line, IF WORDP :TREE [FINISH.UP :TREE STOP], means that if :TREE is a word—that is, an animal name—EXPLORE calls FINISH.UP with the animal name as input and STOPS. If :TREE is not a word, it's a subtree, so EXPLORE follows either its yes branch or its no branch.



Here are two paths EXPLORE can take if its first input is [[IS IT FURRY?] [[DOES IT BARK?] DOG CAT] FROG]:

```
EXPLORE [[IS IT FURRY?] [[DOES IT BARK?] DOG CAT] FROG]
```

The player answers "yes" to IS IT FURRY?

```
EXPLORE [[DOES IT BARK?] DOG CAT]
```

The player answers "no" to DOES IT BARK?

```
EXPLORE "CAT
```

EXPLORE calls FINISH.UP with CAT as input and STOPs

```
EXPLORE STOPs
```

```
EXPLORE STOPs
```

```
EXPLORE [[IS IT FURRY?] [[DOES IT BARK?] DOG CAT] FROG]
```

The player answers "no" to IS IT FURRY?

```
EXPLORE "FROG
```

EXPLORE calls FINISH.UP with FROG as input and STOPs

```
EXPLORE STOPs
```

No matter what path EXPLORE takes, it always ends at an animal name, which it passes to FINISH.UP.

## Guessing and Learning

### Guessing

When EXPLORE calls FINISH.UP, the game is ready to guess that FINISH.UP's input (:BEAST) is your animal. FINISH.UP calls GUESS to do the actual guessing. If GUESS outputs TRUE, the game's guess is right, and BRAG is called. If GUESS outputs FALSE, the game's guess is wrong, and LEARN is called.

```
TO FINISH.UP :BEAST
```

```
IF GUESS :BEAST [BRAG] [LEARN :BEAST [] []]
```

```
END
```

```
TO GUESS :BEAST
```

```
OP YESP (SE [IS IT] A.OR.AN :BEAST [?])
```

```
END
```

```
TO BRAG
```

```
PR [I WIN!]
```

```
PR [I WIN!]
```

```
END
```

### Learning

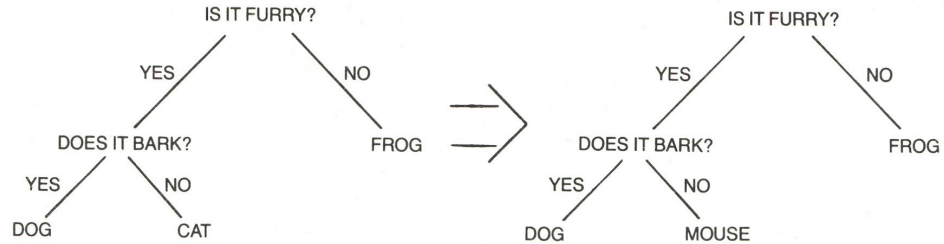
#### *LEARN Adds to the Game's Knowledge*

How does the animal game get smarter? Let's review the imaginary game we played earlier. Our secret animal was "mouse," and the game guessed CAT. Obviously, if the game had guessed "mouse" instead of CAT, it would have won. We might want to change the game so that, from now on, it will guess MOUSE whenever it would have guessed CAT.



## WORDPLAY

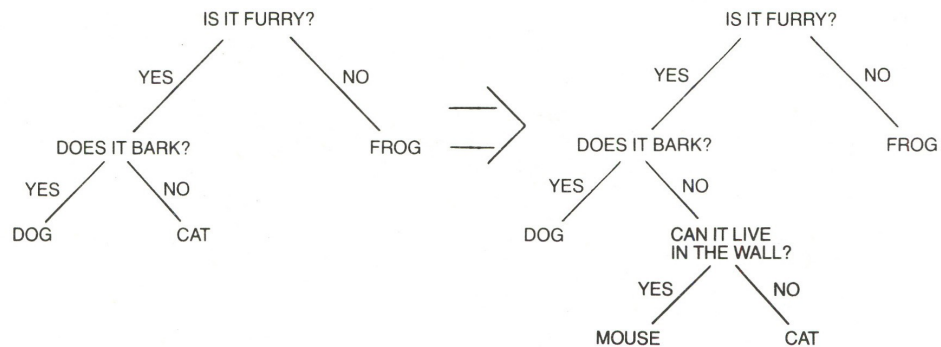
Look at the tree below. To make the game guess MOUSE instead of CAT, we could remove CAT (the wrong guess) from the tree and put MOUSE (the right guess) in its place.



Has the game learned? Not really. We've added a new animal to its knowledge, but we've also subtracted one.

If we want the game's knowledge to include both MOUSE and CAT, we must teach the game a new question, such as CAN IT LIVE IN THE WALL? We also teach it that if a player answers "yes" to the new question, it should guess MOUSE, and if a player answers "no," it should guess CAT.

The next tree shows the result of adding a new animal and a new question to the game's tree. Instead of replacing CAT with MOUSE, we replace CAT with a *new subtree*. The subtree—like all trees—consists of a question (CAN IT LIVE IN THE WALL?), a yes branch (MOUSE), and a no branch (CAT).



### Building a New Subtree

GET.RIGHT.GUESS and GET.NEW.QUESTION get parts for a new subtree.

```

TO GET.RIGHT.GUESS
PR [I GIVE UP. WHAT IS IT?]
OP LAST RL
END
  
```

```

TO GET.NEW.QUESTION
PR [PLEASE TYPE IN A NEW QUESTION]
(PR [WHOSE ANSWER IS 'YES' FOR] :RIGHT.GUESS)
(PR [AND 'NO' FOR] :WRONG.GUESS)
OP RL
END
  
```

*Adding to the Game's "Tree of Knowledge"*

The game's entire "tree of knowledge" is stored in the global variable `BIGTREE`. For the game to get smarter, the new subtree must be added to `:BIGTREE`. `LEARN` and `ALTER` are the main procedures that do this. `ALTER` uses `3LIST`, which outputs a list of its three inputs.

```

TO LEARN :WRONG.GUESS :RIGHT.GUESS :NEW.QUESTION
MAKE "RIGHT.GUESS GET.RIGHT.GUESS
MAKE "NEW.QUESTION GET.NEW.QUESTION
MAKE "BIGTREE ALTER
      :BIGTREE
      :NEW.QUESTION
      :RIGHT.GUESS
      :WRONG.GUESS
END

TO ALTER :TREE :NEW.QUESTION :RIGHT.GUESS :WRONG.GUESS
IF :TREE = :WRONG.GUESS
  [OP 3LIST :NEW.QUESTION :RIGHT.GUESS :WRONG.GUESS]
IF WORDP :TREE
  [OP :TREE]
OP 3LIST (QUESTION :TREE)
      (ALTER YES.BRANCH :TREE :NEW.QUESTION
        :RIGHT.GUESS :WRONG.GUESS)
      (ALTER NO.BRANCH :TREE :NEW.QUESTION
        :RIGHT.GUESS :WRONG.GUESS))
END

```

Let's recall how `LEARN` is called. `EXPLORE` climbs down to an animal name and passes the animal to `FINISH.UP`. `FINISH.UP` calls `GUESS` to guess the animal. If the guess is right, `BRAG` is called. If the guess is wrong, `LEARN` is called.

`LEARN` has three inputs. When it is called, `:WRONG.GUESS` is the animal the game guessed, and `:RIGHT.GUESS` and `:NEW.QUESTION` are empty lists.

`LEARN` calls `GET.RIGHT.GUESS` to get the player's secret animal and stores this animal in `:RIGHT.GUESS`. It calls `GET.NEW.QUESTION` to get the player's new yes-or-no question and stores it in `:NEW.QUESTION`. Then `LEARN` makes `BIGTREE` the output from `ALTER`.

`ALTER`'s four inputs are the game's current "tree of knowledge" and the three parts for the new subtree. `ALTER` looks through the game's current tree, finds the animal the game guessed, and replaces this wrong guess with the new subtree. It then outputs a new, enlarged "tree of knowledge" to `LEARN`.

Here's a sample set of inputs to `ALTER`.

```

:TREE      [[IS IT FURRY?]
            DOES IT BARK?] DOG CAT] FROG]

:NEW.QUESTION  [CAN IT LIVE IN THE WALL?]

:RIGHT.GUESS   MOUSE

:WRONG.GUESS   CAT

```

## WORDPLAY

The following display traces how ALTER works with the preceding inputs. The only input traced is :TREE, since the other inputs are the same each time ALTER is called recursively.

```
ALTER [[IS IT FURRY?] [[DOES IT BARK?] DOG CAT] FROG]
  ALTER [[DOES IT BARK?] DOG CAT]
    ALTER "DOG
    ALTER outputs "DOG
    ALTER "CAT
    ALTER outputs [[CAN IT LIVE IN THE WALL?] MOUSE CAT]
  ALTER outputs [[DOES IT BARK?] DOG [[CAN
    IT LIVE...?] MOUSE CAT]]
  ALTER "FROG
  ALTER outputs "FROG
ALTER outputs [[IS IT FURRY?] [[DOES IT BARK?] DOG [[CAN
  IT LIVE...?] MOUSE CAT] FROG]
```

*Starting the Game*

You begin each session with the animal game by typing ANIMALGAME. This procedure checks whether the game knows anything yet. If no variable named :BIGTREE exists in your workspace, the game knows no questions or animals, so MAKETREE creates a "tree of knowledge" and puts it in :BIGTREE.

PLAY prompts you to think of a secret animal; calls EXPLORE with :BIGTREE as input; and, when the game is over, asks if you'd like to play again.

```
TO ANIMALGAME
IF NOT NAMEP "BIGTREE [MAKETREE]
PLAY
END

TO MAKETREE
MAKE "BIGTREE [[IS IT FURRY?] CAT FROG]
END

TO PLAY
PR []
PR [PICK AN ANIMAL, ANY ANIMAL]
EXPLORE :BIGTREE
IF YESP [DO YOU WANT TO PLAY AGAIN?] [PLAY]
END
```

Remember that every time you play the animal game and it loses, :BIGTREE gets "bigger." And the bigger the game's "tree of knowledge," the smarter the game appears to be.

Since :BIGTREE is a global variable, it remains in your workspace after you've finished a session with the animal game (that is, after you answer "no" to the game's question, DO YOU WANT TO PLAY AGAIN?). If you save this workspace, :BIGTREE will be saved as well. At another session, you could make the game's knowledge even bigger.



If you ever want to *erase* the game's knowledge, stop playing the game and call MAKETREE. MAKETREE causes the game to forget everything it has ever learned.

### *Other Procedures Used by the Game*

All these procedures were mentioned earlier but we did not look at how they work.

The input to A.OR.AN should be an animal name. Its output is the animal name preceded by an appropriate article—either “a” or “an.”

```
TO A.OR.AN :ANIMAL
IF MEMBERP FIRST :ANIMAL [A E I O U] [OP SE "AN :ANIMAL]
OP SE "A :ANIMAL
END
```

YESP and COMPLAIN get a yes-or-no answer to a question. The question is the input to YESP.

```
TO YESP :QUESTION
PR :QUESTION
MAKE "ANS RL
IF NOT OR (EQUALP :ANS [YES]) (EQUALP :ANS [NO])
[COMPLAIN OP YESP :QUESTION]
OP EQUALP :ANS [YES]
END
```

```
TO COMPLAIN
PR [PLEASE ANSWER YES OR NO]
END
```

Here's an example.

```
?PR YESP [IS IT FURRY?]
IS IT FURRY?
SORT OF
PLEASE ANSWER YES OR NO
IS IT FURRY?
YES
TRUE
?
```

3LIST outputs a list of its three inputs.

```
TO 3LIST :ONE :TWO :THREE
OP FPUT :ONE FPUT :TWO FPUT :THREE []
END
```

### SUGGESTIONS

You can play this game with exotic animal names such as armadillo, gnu, gazelle, iguana. You could even use fantastic animals like centaurs or pushme-pullyous. Some people say that it's most fun to play it with the names of your friends!

## PROGRAM LISTING

```

TO QUESTION :TREE
OP FIRST :TREE
END

TO YES.BRANCH :TREE
OP FIRST BF :TREE
END

TO NO.BRANCH :TREE
OP FIRST BF BF :TREE
END

TO EXPLORE :TREE
IF WORDP :TREE [FINISH.UP :TREE STOP]
IF YESP QUESTION :TREE [EXPLORE ►
    YES.BRANCH :TREE] [EXPLORE ►
    NO.BRANCH :TREE]
END

TO FINISH.UP :BEAST
IF GUESS :BEAST [BRAG] [LEARN :BEAST ►
    [] []]
END

TO GUESS :BEAST
OP YESP (SE [IS IT] A.OR.AN :BEAST ►
    [?])
END

TO BRAG
PR [I WIN!]
PR [I WIN!]
END

TO GET.RIGHT.GUESS
PR [I GIVE UP. WHAT IS IT?]
OP LAST RL
END

TO GET.NEW.QUESTION
PR [PLEASE TYPE IN A NEW QUESTION]
(PR [WHOSE ANSWER IS 'YES' FOR] ►
    :RIGHT.GUESS)
(PR [AND 'NO' FOR] :WRONG.GUESS)
OP RL
END

TO LEARN :WRONG.GUESS :RIGHT.GUESS ►
    :NEW.QUESTION
MAKE "RIGHT.GUESS GET.RIGHT.GUESS
MAKE "NEW.QUESTION GET.NEW.QUESTION
MAKE "BIGTREE ALTER :BIGTREE ►
    :NEW.QUESTION :RIGHT.GUESS ►
    :WRONG.GUESS
END

TO ALTER :TREE :NEW.QUESTION ►
    :RIGHT.GUESS :WRONG.GUESS
IF :TREE = :WRONG.GUESS [OP 3LIST ►
    :NEW.QUESTION :RIGHT.GUESS ►
    :WRONG.GUESS]
IF WORDP :TREE [OP :TREE]
OP 3LIST (QUESTION :TREE) (ALTER ►
    YES.BRANCH :TREE :NEW.QUESTION ►
    :RIGHT.GUESS :WRONG.GUESS) (ALTER ►
    NO.BRANCH :TREE :NEW.QUESTION ►
    :RIGHT.GUESS :WRONG.GUESS))
END

TO ANIMALGAME
IF NOT NAMEP "BIGTREE [MAKETREE]
PLAY
END

TO MAKETREE
MAKE "BIGTREE [[IS IT FURRY?] CAT ►
    FROG]
END

TO PLAY
PR []
PR [PICK AN ANIMAL, ANY ANIMAL]
EXPLORE :BIGTREE
IF YESP [DO YOU WANT TO PLAY AGAIN?] ►
    [PLAY]
END

TO A.OR.AN :ANIMAL
IF MEMBERP FIRST :ANIMAL [A E I O U] ►
    [OP SE "AN :ANIMAL]
OP SE "A :ANIMAL
END

```

```

TO YESP :QUESTION
PR :QUESTION
MAKE "ANS RL
IF NOT OR (EQUALP :ANS [YES]) (EQUALP ►
    :ANS [NO]) [COMPLAIN OP YESP ►
    :QUESTION]
OP EQUALP :ANS [YES]
END

```

```

TO COMPLAIN
PR [PLEASE ANSWER YES OR NO]
END

TO 3LIST :ONE :TWO :THREE
OP FPUT :ONE FPUT :TWO FPUT :THREE []
END

MAKE "BIGTREE [[IS IT FURRY?] CAT ►
    FROG]

```

## Dictionary

The idea for this project came about while I was hiking with some friends. During our climb up the mountain, we tried to stump each other by asking the meaning of unusual words. I began to think about developing a dictionary project using Logo.

I wanted to be able to do several things with my dictionary:

- Add a new word and its definition.
- Print the definition of a word.
- Remove a word and its definition.
- Print the entire dictionary.

### *The Dictionary*

My first task was to decide how to store the words. I decided that the dictionary would be a list of entries. Each entry would be a list composed of a word and its definition. Here are two examples.

```
[ICE [FROZEN WATER]]
```

or

```
[HAT [COVERING FOR HEAD]]
```

I named the dictionary `ENTRY.LIST`. Here's how I created it.

```

MAKE "ENTRY.LIST [[EGREGIOUS [CONSPICUOUSLY BAD]]
    [PROSY [COMMONPLACE]]
    [AUTO-DA-FE [BURNING OF A HERETIC]]]

```

### *Using the Dictionary*

When you type `DICTIONARY`, the following is printed on your screen:



**WORDPLAY**

WELCOME TO THE DICTIONARY.

HERE ARE THE COMMANDS:

TYPE A - TO ADD NEW ENTRY  
 TYPE D - TO PRINT DEFINITION OF WORD  
 TYPE P - TO PRINT DICTIONARY  
 TYPE Q - TO QUIT  
 TYPE R - TO REMOVE ENTRY  
 TYPE ? - TO PRINT COMMANDS

TYPE COMMAND.

>

DICTIONARY calls INIT, which checks to see if you already have a dictionary. If you do not, INIT creates one.

```
TO DICTIONARY
INIT
PR [WELCOME TO THE DICTIONARY.]
PR []
PR [HERE ARE THE COMMANDS:]
DO.CHOICE "?"
END
```

```
TO INIT
CT TS
IF NOT NAMEP "ENTRY.LIST [MAKE "ENTRY.LIST
  [[EGREGIOUS [CONSPICUOUSLY BAD]]
  [PROSY [COMMONPLACE]]
  [[AUTO-DA-FE] [BURNING OF A HERETIC]]]]
END
```

DO.CHOICE has the job of figuring out whether the character you type matches one of the expected commands. If there is no match or if you type ?, DO.CHOICE prints the list of possible choices.

```
TO DO.CHOICE :LTR
PR []
IF EQUALP "A :LTR [ADD.ENTRY]
IF EQUALP "D :LTR [PRINT.DEFINITION]
IF EQUALP "P :LTR [PRINT.DICTIONARY]
IF EQUALP "Q :LTR [STOP]
IF EQUALP "R :LTR [REMOVE.ENTRY]
IF NOT MEMBERP :LTR [A D P Q R] [PRINT.CHOICES]
PR []
PR [TYPE COMMAND.]
TYPE ">"
MAKE "LTR RC
PR :LTR
DO.CHOICE :LTR
END
```

```

TO PRINT.CHOICES
PR [TYPE A - TO ADD NEW ENTRY]
PR [TYPE D - TO PRINT DEFINITION OF WORD]
PR [TYPE P - TO PRINT DICTIONARY]
PR [TYPE Q - TO QUIT]
PR [TYPE R - TO REMOVE ENTRY]
PR [TYPE ? - TO PRINT COMMANDS]
END

```

### *Adding a New Word and Definition*

To add a word, you type A while running DICTIONARY. Here's an example of what happens.

```

TYPE NEW WORD.
FLUMP
TYPE DEFINITION OF NEW WORD.
DROP OR MOVE HEAVILY

TYPE COMMAND
>

```

If you try to add a word that is already in the dictionary, this happens:

```

TYPE NEW WORD
EGREGIOUS
EGREGIOUS IS ALREADY IN DICTIONARY.

```

ADD.ENTRY is the procedure that lets you add a new entry to the dictionary.

```

TO ADD.ENTRY
PR [TYPE NEW WORD.]
ADD.ENTRY1 FIRST RL
END

```

ADD.ENTRY1 calls GET.ENTRY to see if the word you want to add is already in the dictionary. If the word is not in the dictionary, then the word and its definition become a new entry.

```

TO ADD.ENTRY1 :WRD
IF NOT EMPTY GET.ENTRY :WRD :ENTRY.LIST
  [PR SE :WRD [IS ALREADY IN DICTIONARY.] STOP]
PR [TYPE DEFINITION OF NEW WORD.]
MAKE.ENTRY LIST :WRD RL
END

```

GET.ENTRY has the task of finding an entry in the dictionary. It does this by attempting to match an input word with the first word in each entry.

```

TO GET.ENTRY :WRD :LST
IF EMPTY :LST [OP []]
IF EQUALP :WRD FIRST :LST [OP FIRST :LST]
OP GET.ENTRY :WRD BF :LST
END

```

**WORDPLAY**

MAKE.ENTRY adds a new entry to the dictionary.

```
TO MAKE.ENTRY :NEW.ENTRY
MAKE "ENTRY.LIST FPUT :NEW.ENTRY :ENTRY.LIST
END
```

***Printing the Definition of a Word***

This is what happens when you type D.

```
TYPE WORD WHOSE DEFINITION
YOU WANT PRINTED.
EGREGIOUS
[CONSPICUOUSLY BAD]
```

PRINT.DEFINITION calls PRINT.DEF1 to print out the definition of a word.

```
TO PRINT.DEFINITION
PR [TYPE WORD WHOSE DEFINITION]
PR [YOU WANT PRINTED.]
PRINT.DEF1 FIRST RL
END
```

```
TO PRINT.DEF1 :WRD
PRINT.DEF2 :WRD GET.ENTRY :WRD :ENTRY.LIST
END
```

PRINT.DEF1 then calls PRINT.DEF2 with the word to be defined and its entry in the dictionary. If the entry is in the dictionary, PRINT.DEF2 prints the definition.

```
TO PRINT.DEF2 :WRD :LST
IF EMPTY? :LST [PR SE :WRD [IS NOT IN DICTIONARY.] STOP]
PR BF :LST
END
```

***Removing an Entry from the Dictionary***

To remove an entry, you type R. Here is an example.

```
TYPE WORD
YOU WANT TO REMOVE.
FLUMP
```

REMOVE.ENTRY uses REMOVE to output a dictionary, minus the unwanted entry.

```
TO REMOVE.ENTRY
PR [TYPE WORD]
PR [YOU WANT TO REMOVE.]
MAKE "ENTRY.LIST REMOVE FIRST RL :ENTRY.LIST
END
```



```

TO REMOVE :WRD :LST
IF EMPTY :LST [PR SE :WRD [IS NOT IN DICTIONARY.] OP []]
IF EQUALP :WRD FIRST FIRST :LST [OP BF :LST]
OP FPUT FIRST :LST REMOVE :WRD BF :LST
END

```

### *Printing the Dictionary*

Here's what happens when you type P. I've added some words that I thought were interesting to the dictionary.

```

IMPUISSANT
[WEAK; IMPOTENT]

```

```

ACCRETIVE
[ADDING IN GROWTH]

```

```

DENTILOQUY
[THE ACT OR HABIT OF SPEAKING WITH TEETH CLOSED]

```

```

CENOSITY
[FILTHINESS; SQUALOR]

```

```

DELIQUESCE
[TO MELT AWAY]

```

```

FETOR
[STRONG OFFENSIVE SMELL]

```

```

BRUMAL
[INDICATIVE OF OR OCCURRING IN WINTER]

```

```

**TYPE ANY CHARACTER
TO SEE MORE**

```

**Note:** At this point you press any key to see the next seven (or remaining) entries.

```

EGREGIOUS
[CONSPICUOUSLY BAD]

```

```

PROSY
[COMMONPLACE]

```

```

AUTO - DA - FE
[BURNING OF A HERETIC]

```

The procedures PRINT.DICTIONARY, FORMAT, and PRINT.ENTRY work together to print ENTRY.LIST in an easy-to-read format. There is room on the screen for seven entries. FORMAT counts the number of entries. When the screen is full, FORMAT pauses and waits until you type any character before printing the next seven or remaining entries.

## WORDPLAY

```

TO PRINT.DICTIONARY
FORMAT 0 :ENTRY.LIST
END

```

```

TO FORMAT :CTR :LST
IF EMPTY :LST [STOP]
IF :CTR = 7
  [PR [**TYPE ANY CHARACTER]
  PR [TO SEE MORE**] PR RC]
PRINT.ENTRY FIRST :LST
FORMAT IF :CTR = 7 [1] [:CTR + 1] BF :LST
END

```

```

TO PRINT.ENTRY :ENTRY
PR FIRST :ENTRY
PR BF :ENTRY
PR []
END

```

## PROGRAM LISTING

```

TO DICTIONARY
INIT
PR [WELCOME TO THE DICTIONARY.]
PR []
PR [HERE ARE THE COMMANDS:]
DO.CHOICE "?"
END

```

```

TO INIT
CT TS
IF NOT NAMEP "ENTRY.LIST [MAKE ►
  "ENTRY.LIST [[EGREGIOUS ►
  [CONSPICUOUSLY BAD]] [PROSY
  [COMMONPLACE]] [[AUTO-DA-FE] ►
  [BURNING OF A HERETIC]]]]
END

```

```

TO DO.CHOICE :LTR
PR []
IF EQUALP "A :LTR [ADD.ENTRY]
IF EQUALP "D :LTR [PRINT.DEFINITION]
IF EQUALP "P :LTR [PRINT.DICTIONARY]
IF EQUALP "Q :LTR [STOP]
IF EQUALP "R :LTR [REMOVE.ENTRY]
IF NOT MEMBERP :LTR [A D P Q R] ►
  [PRINT.CHOICES]
PR []
PR [TYPE COMMAND.]
TYPE ">"
MAKE "LTR RC
PR :LTR
DO.CHOICE :LTR
END

```

```

TO PRINT.CHOICES
PR [TYPE A - TO ADD NEW ENTRY]
PR [TYPE D - TO PRINT DEFINITION OF ►
  WORD]
PR [TYPE P - TO PRINT DICTIONARY]
PR [TYPE Q - TO QUIT]
PR [TYPE R - TO REMOVE ENTRY]
PR [TYPE ? - TO PRINT COMMANDS]
END

```

```

TO ADD.ENTRY
PR [TYPE NEW WORD.]
ADD.ENTRY1 FIRST RL
END

```

```

TO ADD.ENTRY1 :WRD
IF NOT EMPTY GET.ENTRY :WRD ►
  :ENTRY.LIST [PR SE :WRD [IS ►
  ALREADY IN DICTIONARY.] STOP]
PR [TYPE DEFINITION OF NEW WORD.]
MAKE.ENTRY LIST :WRD RL
END

```

```

TO GET.ENTRY :WRD :LST
IF EMPTY :LST [OP []]
IF EQUALP :WRD FIRST FIRST :LST [OP ►
  FIRST :LST]
OP GET.ENTRY :WRD BF :LST
END

```



```

TO MAKE.ENTRY :NEW.ENTRY
MAKE "ENTRY.LIST FPUT :NEW.ENTRY ►
:ENTRY.LIST
END

TO PRINT.DEFINITION
PR [TYPE WORD WHOSE DEFINITION]
PR [YOU WANT PRINTED.]
PRINT.DEF1 FIRST RL
END

TO PRINT.DEF1 :WRD
PRINT.DEF2 :WRD GET.ENTRY :WRD ►
:ENTRY.LIST
END

TO PRINT.DEF2 :WRD :LST
IF EMPTY :LST [PR SE :WRD [IS NOT IN ►
DICTIONARY.] STOP]
PR BF :LST
END

TO REMOVE.ENTRY
PR [TYPE WORD]
PR [YOU WANT TO REMOVE.]
MAKE "ENTRY.LIST REMOVE FIRST RL ►
:ENTRY.LIST
END

TO REMOVE :WRD :LST
IF EMPTY :LST [PR SE :WRD [IS NOT IN ►
DICTIONARY.] OP []]
IF EQUALP :WRD FIRST FIRST :LST [OP BF ►
:LST]
OP FPUT FIRST :LST REMOVE :WRD BF :LST
END

```

```

TO PRINT.DICTIONARY
FORMAT 0 :ENTRY.LIST
END

TO FORMAT :CTR :LST
IF EMPTY :LST [STOP]
IF :CTR = 7 [PR [**TYPE ANY CHARACTER] ►
PR [TO SEE MORE**] PR RC]
PRINT.ENTRY FIRST :LST
FORMAT IF :CTR = 7 [1] [:CTR + 1] BF ►
:LST
END

TO PRINT.ENTRY :ENTRY
PR FIRST :ENTRY
PR BF :ENTRY
PR []
END

MAKE "ENTRY.LIST [[IMPUISSANT [WEAK; ►
IMPOTENT]] [ACCRETIVE [ADDING IN ►
GROWTH]] [DENTILOQUY [THE ACT OR ►
HABIT OF SPEAKING WITH TEETH ►
CLOSED]] [CENOSITY [FILTHINESS; ►
SQUALOR]] [DELIQUESCE [TO MELT ►
AWAY]] [FETOR [STRONG OFFENSIVE ►
SMELL]] [BRUMAL [INDICATIVE OF OR ►
OCCURRING IN WINTER]] [EGREGIOUS ►
[CONSPICUOUSLY BAD]] [PROSY ►
[COMMONPLACE]] [[AUTO - DA - FE] ►
[BURNING OF A HERETIC]]]

```

## Hangman

HANGMAN is based on the popular two-person pencil-and-paper game in which one player thinks up a secret word and the other player tries to discover what the word is by guessing what letters are in the word. A gallows is drawn, and for each incorrect guess, part of a stick figure is added to the drawing. The player who is guessing wins the game by guessing the entire word before the stick figure is completed.

---

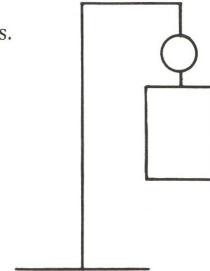
By Brian Harvey.

## WORDPLAY

In this version, the program chooses the secret word and you do the guessing. At each turn, you can guess either a single letter or the entire word.

Here is a picture of a game in progress.

```
-A--E-
GUESSES: E T A O I
YOUR GUESS?
```



The secret word is shown as -A--E-. This means that it has six letters, two of which have been guessed. You have made five guesses. A and E were correct. The others, T, O, and I, were wrong. Because of these three wrong guesses, the program has drawn the head, neck, and body of the person being hanged. If you make more wrong guesses, the program will draw the person's arms and legs.

I like this program because it combines text processing with graphics. The top-level procedure divides the program into two parts: setting up and playing the game.

```
TO HANGMAN
  SETUP
  PLAY
END
```

*Setting Up*

SETUP has two jobs; it gives initial values to certain variables, including the secret word, and it draws the gallows.

```
TO SETUP
  MAKE "MYWORD PICKWORD
  MAKE "GUESSES []
  MAKE "WON "FALSE
  MAKE "SPACES "
  REPEAT 18 [MAKE "SPACES WORD :SPACES CHAR 32]
  MAKE "GOTTEN 0
  MAKE "TRIES 7
  CT
  GALLOWS
END
```

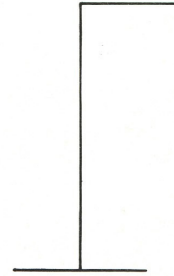
SETUP uses two main subprocedures, one to pick the secret word and one to draw the gallows. PICKWORD outputs the secret word, which SETUP remembers in the global variable MYWORD. To choose the word from a list of possible words, PICKWORD uses the procedures PICK and ITEM, which appear as examples in the *Atari Logo Reference Manual*.

```
TO PICKWORD
  OP PICK [POTSTICKER COMPUTER IRAQ GAZEBO THRUSH STYLE FOILED
           SWARM ZEBRA AWFUL WILY YELLOW BARKED STOIC]
END
```

GALLOWS positions a turtle for drawing the gallows, sets the pen down, and uses GALL1 to do the actual drawing. The reason to make GALL1 a subprocedure is that it will be used again, with the eraser down, to erase the gallows if you win by guessing the word.

```
TO GALLOWS
  TELL [0 1 2 3]
  CS HT
  TELL 0
  PU SETPOS [-40 -60]
  RT 90
  PD
  GALL1
END
```

```
TO GALL1
  FD 80
  BK 40
  LT 90
  FD 170
  RT 90
  FD 60
  RT 90
  FD 20
END
```



### *Variables Created by SETUP*

The variable MYWORD is one of several that are used throughout the hangman program to keep track of the progress of the game. For example, the program must remember what letters have been guessed and how many wrong guesses are allowed before you lose. Several of these variables are given their initial values by SETUP.

MYWORD	The secret word.
GUESSES	A list of the letters you have guessed.
WON	TRUE if you win by guessing the word or the last missing letter in it.
SPACES	A word of eighteen spaces, which is typed to erase messages from the program in the text part of the screen.
GOTTEN	The number of letters in the secret word that you have guessed correctly. (If a letter occurs more than once in the secret word, the number of letters guessed correctly may be more than the number of correct guesses you have made, because one correct guess may reveal several letters in the word.)
TRIES	The number of incorrect guesses remaining before you lose.

### *Playing the Game*

The central part of the hangman program is the procedure PLAY and its subprocedure GETGUESS, which is called each time you make a guess.



**WORDPLAY**

```

TO PLAY
IF :TRIES=0 [LOSE STOP]
GETGUESS
IF :WON [SETCURSOR [0 23] STOP]
PLAY
END

TO GETGUESS
DISPLAY
MAKE "GUESS FIRST RL
CLEARMESSAGE
IF (COUNT :GUESS) > 1 [TESTWORD STOP]
IF MEMBERP :GUESS :GUESSES [ALREADY GETGUESS STOP]
MAKE "GUESSES SE :GUESSES :GUESS
IF MEMP :GUESS :MYWORD [FIXGOT :GUESS :MYWORD] [BADTRY]
IF EQUALP :GOTTEN COUNT :MYWORD [WIN]
END

```

PLAY calls GETGUESS repeatedly, checking between times to see if you've won (the variable WON made TRUE) or lost (no more TRIES left). GETGUESS uses several subprocedures to display the current state of the game, read a guess from the keyboard, and test the guess. A guess can be either a single letter or the entire word. These cases are distinguished by checking the COUNT of the guess; if it's more than one letter, the procedure TESTWORD is used to compare the guess to the secret word. Otherwise, the program checks if you have already guessed the letter; if not, it checks to see if the guessed letter is actually in the word. If the letter is in the word, FIXGOT is called to update the number of letters correctly guessed. If not, BADTRY draws another piece of the body under the gallows.

***Keeping Track of the Text Screen***

The text part of the screen in the middle of a game might look like this:

-A-E--	YOU GUESSED THAT!
GUESSES: E T A	
YOUR GUESS? _	

In the top left corner is the display of the secret word, with some letters already guessed and the others indicated by hyphens. In the top right corner is the *message area*. You have just repeated a guess already made, and the program has complained about it. The next line shows the list of letters already guessed. The third line invites you to make another guess, and the cursor is positioned for reading that guess.

The message area is maintained by the procedure SAY. Two subprocedures of GETGUESS show simple examples of how SAY is used:

```

TO SAY :STUFF
SETCURSOR [19 20]
TYPE :STUFF
END

```

```

TO ALREADY
SAY [YOU GUESSED THAT!]
END

```

```

TO CLEARMESSAGE
SAY :SPACES
END

```

(The underlining in this listing represents inverse-video characters on the screen.) The CLEARMESSAGE procedure types spaces into the message area, erasing any leftover messages. The procedure ALREADY is called by GETGUESS if you repeat a previous guess.

The rest of the text screen, apart from the message area, is maintained by the DISPLAY procedure:

```

TO DISPLAY
SETCURSOR [0 20]
DISWORD :MYWORD
SETCURSOR [0 21]
TYPE "GUESSES:
SETCURSOR [9 21]
TYPE :GUESSES
SETCURSOR [0 22]
TYPE [YOUR GUESS?]
SETCURSOR [12 22]
END

```

For each letter of the secret word, DISWORD looks in the list of letters already guessed. If this letter has been guessed, DISWORD types it. If not, DISWORD types a hyphen.

```

TO DISWORD :WORD
IF EMPTY :WORD [STOP]
IF MEMBERP FIRST :WORD :GUESSES [TYPE FIRST :WORD] [TYPE "-"]
DISWORD BF :WORD
END

```

### *When You Guess a Letter*

When you guess a letter (that hasn't been guessed already), GETGUESS calls either FIXGOT or BADTRY depending on whether the guess is correct or incorrect. To test the correctness of the guess, GETGUESS uses MEMP, which is like the primitive MEMBERP except that it checks whether a letter is an element of a word, instead of whether a word is an element of a list.

```

TO MEMP :LETTER :WORD
IF EMPTY :WORD [OP "FALSE]
IF EQUALP :LETTER FIRST :WORD [OP "TRUE]
OP MEMP :LETTER BF :WORD
END

```

(Actually, MEMP would work equally well testing for membership in a list, like MEMBERP, but we need it only to check for membership in a word.)

If the guess is correct, the task of FIXGOT is to calculate a new value



## WORDPLAY

for the variable GOTTEN, which counts the number of correctly guessed letters in the secret word. We can't just add 1 to GOTTEN, because the letter you guessed may appear more than once in the secret word. For example, if the secret word is "thrush" and you guess H, FIXGOT must add 2 to GOTTEN. So FIXGOT must examine each letter of the secret word.

```
TO FIXGOT :GUESS :WORD
IF EMPTY? :WORD [STOP]
IF EQUALP :GUESS FIRST :WORD [MAKE "GOTTEN :GOTTEN+1]
FIXGOT :GUESS BF :WORD
END
```

Note that FIXGOT does not actually display the newly guessed letters on the screen. This will be done by DISPLAY the next time through GETGUESS.

*What Happens on a Wrong Guess*

If the guess is incorrect, BADTRY is called to count down the number of turns until you lose and to draw part of the body under the gallows:

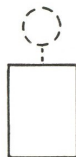
```
TO BADTRY
RUN SE WORD "DRAW :TRIES []
MAKE "TRIES :TRIES-1
END
```

The RUN command is used to select a subprocedure to draw the appropriate part of the body, based on the number of tries remaining. For example, the variable TRIES is initially 7, and the procedure DRAW7 draws a head. DRAW6 draws the neck, DRAW5 the torso, DRAW4 and DRAW3 the arms, and DRAW2 and DRAW1 the legs:

```
TO DRAW7
PU
SETPOS [60 90]
SETH 105
PD
REPEAT 12 [FD 6 RT 30]
RT 75
END
```

```
TO DRAW6
PU
SETPOS [60 66]
SETH 180
PD
FD 10
END
```

```
TO DRAW5
PU
SETPOS [40 56]
SETH 90
PD
REPEAT 2 [FD 40 RT 90 FD 60 RT 90]
END
```





```

TO DRAW4
PU
SETPOS [40 40]
SETH -45
PD
ARM
END

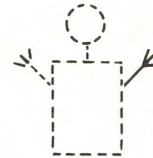
```



```

TO DRAW3
PU
SETPOS [80 40]
SETH 45
PD
ARM
END

```



```

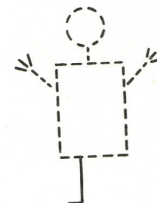
TO ARM
FD 30
BK 14
LT 25
FD 14
BK 14
RT 50
FD 14
END

```

```

TO DRAW2
PU
SETPOS [52 -4]
SETH 180
PD
FD 30
RT 90
FD 8
END

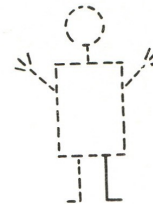
```



```

TO DRAW1
PU
SETPOS [68 -4]
SETH 180
PD
FD 30
LT 90
FD 8
END

```



None of the procedures DRAW1, DRAW2, and so forth, assume that the turtle is at any particular position. This is because if you win, the program will erase the gallows and then finish drawing in the body, so any of these

## WORDPLAY

procedures might be called with the turtle at the end of the gallows, rather than at the end of the previous body part.

*When You Guess a Word*

We have looked at the procedures that deal with a guess of a single letter. You may also guess the entire word; if so, the GETGUESS procedure calls TESTWORD.

```
TO TESTWORD
IF EQUALP :GUESS :MYWORD [WIN] [SAY [NOPE!]] BADTRY]
END
```

An incorrect guess of the entire word is handled by BADTRY, just like an incorrect guess of a letter. But if you guess the entire word correctly, there is no need to call FIXGOT. We can simply call WIN, because you have won the game.

*When You Lose the Game*

We have now looked at all the procedures involved in playing the game, up to the point of winning or losing. The case of losing is easier to understand. You lose by running out of tries. This means that the entire body has already been drawn.

```
TO LOSE
SAY [YOU LOSE!! SORRY.]
SETCURSOR [0 20]
TYPE :MYWORD
SETCURSOR [0 23]
EYES
FROWN
END
```

```
TO EYES
PU
SETPOS [52 82]
SETH 90
PD
FD 4
PU
FD 6
PD
FD 4
END
```



```
TO FROWN
PU
SETPOS [66 72]
SETH -9
MOUTH
END
```



```

TO MOUTH
PD
LT 18
REPEAT 8 [FD 2 LT 18]
END

```

The program tells you what the secret word was, moves the cursor down to the last screen line, and fills in the already-drawn head with a frowning face. When the program stops, Logo will print its prompt on the last line without obscuring what is written in the text area. (LOSE is called only by PLAY, which then stops, returning to HANGMAN, which stops. So when LOSE stops, the entire program is done.)

### *When You Win the Game*

What if you win? In this case, the body is not yet entirely drawn. We want to erase the gallows, finish drawing the body, notify the winner, and stop the program.

```

TO WIN
SETCURSOR [0 20]
DISWORD :MYWORD
SAY [YOU WIN!!!]
MAKE "WON "TRUE
UNGALL
FINISH :TRIES
EYES
SMILE
END

```

```

TO UNGALL
PU
SETPOS [-40 -60]
SETH 90
PE
GALL1
END

```

```

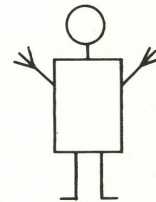
TO FINISH :NUM
IF :NUM=0 [STOP]
RUN SE WORD "DRAW :NUM []
FINISH :NUM-1
END

```

```

TO SMILE
PU
SETPOS [54 76]
SETH 171
MOUTH
END

```



UNGALL is like GALLOWS except that it draws the gallows in PE (penerase). FINISH calls each of the yet-undone drawing procedures (DRAW1, etc.) to



## WORDPLAY

finish drawing the body. And SMILE draws the same mouth as FROWN, but right-side up.

Unlike LOSE, the WIN procedure can be called from two places in the program: TESTWORD and GETGUESS. Because these places are deeper in the chain of subprocedures, we must set the variable WON so that the PLAY procedure can test it, to know when to stop the game program.

*Utilities*

To complete the listing of procedures used in this project, here are the utility procedures PICK and ITEM:

```

TO PICK :LIST
OP ITEM (1 + RANDOM COUNT :LIST) :LIST
END

TO ITEM :N :LIST
IF :N=1 [OP FIRST :LIST]
OP ITEM :N-1 BF :LIST
END

```

## PROGRAM LISTING

TO HANGMAN	TO GALL1
SETUP	FD 80
PLAY	BK 40
END	LT 90
	FD 170
TO SETUP	RT 90
MAKE "MYWORD PICKWORD	FD 60
MAKE "GUESSES []	RT 90
MAKE "WON "FALSE	FD 20
MAKE "SPACES "	END
REPEAT 18 [MAKE "SPACES WORD :SPACES ►	
CHAR 32]	TO PLAY
MAKE "GOTTEN 0	IF :TRIES=0 [LOSE STOP]
MAKE "TRIES 7	GETGUESS
CT	IF :WON [SETCURSOR [0 23] STOP]
GALLOWS	PLAY
END	END
TO GALLOWS	TO GETGUESS
TELL [0 1 2 3]	DISPLAY
CS HT	MAKE "GUESS FIRST RL
TELL 0	CLEARMESSAGE
PU SETPOS [-40 -60]	IF (COUNT :GUESS) > 1 [TESTWORD STOP]
RT 90	IF MEMBERP :GUESS :GUESSES [ALREADY ►
PD	GETGUESS STOP]
GALL1	MAKE "GUESSES SE :GUESSES :GUESS
END	IF MEMP :GUESS :MYWORD [FIXGOT :GUESS ►
	:MYWORD] [BADTRY]
	IF EQUALP :GOTTEN COUNT :MYWORD [WIN]
	END

# HANGMAN

37

```
TO PICKWORD
OP PICK [POTSTICKER COMPUTER IRAQ ►
    GAZEBO THRUSH STYLE FOILED SWARM ►
    ZEBRA AWFUL WILY YELLOW BARKED ►
    STOIC]
END
```

```
TO SAY :STUFF
SETCURSOR [19 20]
TYPE :STUFF
END
```

```
TO ALREADY
SAY [YOU GUESSED THAT!]
END
```

```
TO CLEARMESSAGE
SAY :SPACES
END
```

```
TO DISPLAY
SETCURSOR [0 20]
DISWORD :MYWORD
SETCURSOR [0 21]
TYPE "GUESSES:
SETCURSOR [9 21]
TYPE :GUESSES
SETCURSOR [0 22]
TYPE [YOUR GUESS?]
SETCURSOR [12 22]
END
```

```
TO DISWORD :WORD
IF EMPTY :WORD [STOP]
IF MEMBERP FIRST :WORD :GUESSES [TYPE ►
    FIRST :WORD] [TYPE "-"]
DISWORD BF :WORD
END
```

```
TO MEMP :LETTER :WORD
IF EMPTY :WORD [OP "FALSE]
IF EQUALP :LETTER FIRST :WORD [OP ►
    "TRUE]
OP MEMP :LETTER BF :WORD
END
```

```
TO FIXGOT :GUESS :WORD
IF EMPTY :WORD [STOP]
IF EQUALP :GUESS FIRST :WORD [MAKE ►
    "GOTTEN :GOTTEN+1]
FIXGOT :GUESS BF :WORD
END
```

```
TO BADTRY
RUN SE WORD "DRAW :TRIES []
```

```
MAKE "TRIES :TRIES-1
END
```

```
TO DRAW7
PU
SETPOS [60 90]
SETH 105
PD
REPEAT 12 [FD 6 RT 30]
RT 75
END
```

```
TO DRAW6
PU
SETPOS [60 66]
SETH 180
PD
FD 10
END
```

```
TO DRAW5
PU
SETPOS [40 56]
SETH 90
PD
REPEAT 2 [FD 40 RT 90 FD 60 RT 90]
END
```

```
TO DRAW4
PU
SETPOS [40 40]
SETH -45
PD
ARM
END
```

```
TO DRAW3
PU
SETPOS [80 40]
SETH 45
PD
ARM
END
```

```
TO ARM
FD 30
BK 14
LT 25
FD 14
BK 14
RT 50
FD 14
END
```

```

TO DRAW2
PU
SETPOS [52 -4]
SETH 180
PD
FD 30
RT 90
FD 8
END

```

```

TO DRAW1
PU
SETPOS [68 -4]
SETH 180
PD
FD 30
LT 90
FD 8
END

```

```

TO TESTWORD
IF EQUALP :GUESS :MYWORD [WIN] [SAY ►
  [NOPE!] BADTRY]
END

```

```

TO LOSE
SAY [YOU LOSE!! SORRY.]
SETCURSOR [0 20]
TYPE :MYWORD
SETCURSOR [0 23]
EYES
FROWN
END

```

```

TO EYES
PU
SETPOS [52 82]
SETH 90
PD
FD 4
PU
FD 6
PD
FD 4
END

```

```

TO FROWN
PU
SETPOS [66 72]
SETH -9
MOUTH
END

```

```

TO MOUTH
PD
LT 18
REPEAT 8 [FD 2 LT 18]
END

```

```

TO WIN
SETCURSOR [0 20]
DISWORD :MYWORD
SAY [YOU WIN!!!]
MAKE "WON "TRUE
UNGALL
FINISH :TRIES
EYES
SMILE
END

```

```

TO UNGALL
PU
SETPOS [-40 -60]
SETH 90
PE
GALL1
END

```

```

TO FINISH :NUM
IF :NUM=0 [STOP]
RUN SE WORD "DRAW :NUM []
FINISH :NUM-1
END

```

```

TO SMILE
PU
SETPOS [54 76]
SETH 171
MOUTH
END

```

```

TO PICK :LIST
OP ITEM (1 + RANDOM COUNT :LIST) :LIST
END

```

```

TO ITEM :N :LIST
IF :N=1 [OP FIRST :LIST]
OP ITEM :N-1 BF :LIST
END

```



## Math: A Sentence Generator

When we think of computers making up sentences, we most often think of them making up English or French sentences. We rarely think of them making up math sentences. This project is about developing a math sentence generator. It is set in the context of developing an interactive program. A sentence is made up in the form  $3 + X = 5$  and the user is asked WHAT IS X?.

The first example involves only addition sentences. Then the program is modified to include multiplication, subtraction, and division. Later the program is changed once more to vary the form of the math sentences and keep track of the number of times the user responds to the same question.

I boldface what the user types.

```
?MATH
6 + X = 7
WHAT IS X? 1
RIGHT
```

```
7 + X = 16
WHAT IS X? 5
NOPE, X IS 9
```

```
7 + X = 10
WHAT IS X? 3
RIGHT
```

As the example shows, MATH makes addition sentences of the form  $2 + X = 3$  and not of the form  $X + 2 = 3$ . Later we will change MATH so that it uses both forms.

MATH randomly chooses two of the integers to be used in the math sentence. ADD then presents the addition problem and checks on your answer. The numbers MATH chooses are less than ten, but you can easily adjust the procedure and make the numbers larger.

```
TO MATH
ADD RANDOM 10 RANDOM 10
PR []
MATH
END

TO ADD :NUM1 :NUM2
PR (SE :NUM1 [+ X =] :NUM1 + :NUM2)
TYPE SE [WHAT IS X?] "
IF :NUM2 = FIRST RL [PR [RIGHT] STOP]
PR SE [NOPE, X IS] :NUM2
END
```

In the addition sentences, the value of X is :NUM2, which is the second input to ADD. The sum of the two inputs is computed by ADD.

There are different ways to expand this program. You could design the

## WORDPLAY

program so that it gives you three chances to get the answer right. You could expand the program so that it gives you problems in subtraction, division, and multiplication. You could make it keep track of the number of problems you do and the number you respond correctly to. You might decide to help the user. Some of these suggestions are explored in the next section.

*Making MATH Subtract, Multiply, and Divide*

One way to extend MATH is to make three more procedures, SUBTRACT, MULTIPLY, and DIVIDE.

```
TO SUBTRACT :NUM1 :NUM2
PR (SE :NUM1 [- X =] :NUM1 - :NUM2)
TYPE SE [WHAT IS X?] "
IF :NUM2 = FIRST RL [PR [RIGHT] STOP]
PR SE [NOPE, X IS] :NUM2
END
```

```
TO MULTIPLY :NUM1 :NUM2
PR (SE :NUM1 [* X =] :NUM1 * :NUM2)
TYPE SE [WHAT IS X?] "
IF :NUM2 = FIRST RL [PR [RIGHT] STOP]
PR SE [NOPE, X IS] :NUM2
END
```

```
TO DIVIDE :NUM1 :NUM2
PR (SE :NUM1 [/ X =] :NUM1 / :NUM2)
TYPE SE [WHAT IS X?] "
IF :NUM2 = FIRST RL [PR [RIGHT] STOP]
PR SE [NOPE, X IS] :NUM2
END
```

Try these procedures to see if there are any bugs. Modifying MATH is a good way to try these new procedures.

```
TO MATH
ADD RANDOM 10 RANDOM 10
SUBTRACT RANDOM 10 RANDOM 10
MULTIPLY RANDOM 10 RANDOM 10
DIVIDE RANDOM 10 RANDOM 10
PR []
MATH
END
```

**MATH**

```
4 + X = 7
WHAT IS X? 3
RIGHT
3 - X = 4
WHAT IS X? 1
NOPE, X IS -1
```

```
6 * X = 18
WHAT IS X? 3
RIGHT
3 / X = 1
WHAT IS X? 3
RIGHT
```



What do you think? The program seems to work, but there are some possible problems. For example, in the subtraction sentences  $X$  might have a negative value. Perhaps you want to use this program without negative numbers for answers. We can adjust `SUBTRACT` so that the value of  $X$  is always positive.

Notice that the sentences are of the form  $3 - X = 2$ . The form  $X - 3 = 4$  might be easier to solve, and so you might want to make sentences in that form.

There is a potential bug with multiplication and division. For example, division by 0 will cause Logo to stop the program and print out an error message. Attempts to divide by 0 must be prevented. One way to make sure of this is to add one to the random number used as `DIVIDE`'s second input. Multiplication by 0 can cause a different sort of problem when you try to figure out what  $0 * X$  is.

Although the preceding examples do not show  $X$  being a fractional number like .5, it is possible. You might want to guard against that happening. Since the sentences are generated by the program, we can make sure that the computation is performed so that  $X$  is always a whole number.

In the next section `MATH` is extended to include some of these ideas. The procedures are rewritten. A new procedure is introduced called `ANSWER`. It is used by `ADD`, `MULTIPLY`, `SUBTRACT`, and `DIVIDE` to print out the sentence and get the user's response to what  $X$  is.

### *Extending* MATH

In this section, the first extensions to `MATH` guard against multiplication or division by 0 and give the user three chances to figure out what  $X$  is. All math sentences are still written in the form  $3 + X = 5$  and expect integer answers. The program generates two random numbers and then computes a third. Here is an example of the program in action.

?MATH

HERE ARE SOME MATH PROBLEMS.

$8 + X = 13$

WHAT IS  $X$ ?

Now if you type 5, Logo responds:

RIGHT ON

If you type anything else, Logo responds:

TRY AGAIN

$8 + X = 13$

WHAT IS  $X$ ?

You are given three tries to get the answer. If you are still wrong, Logo responds:

NOPE,  $X$  IS 5



## WORDPLAY

MATH has MATH1 present sentences in subtraction, multiplication, and division as well as addition.

```
TO MATH
TS
CT
PR [HERE ARE SOME MATH PROBLEMS.]
MATH1
END
```

```
TO MATH1
ADD RANDOM 11 RANDOM 11
SUBTRACT RANDOM 11 RANDOM 11
MULTIPLY 1 + RANDOM 10 1 + RANDOM 10
DIVIDE 1 + RANDOM 10 1 + RANDOM 10
PR [] WAIT 60
MATH1
END
```

After ADD computes :RESULT, ANSWER takes over the job of printing out the sentence and checking the user's response.

```
TO ADD :NUM1 :NUM2
MAKE "RESULT :NUM1 + :NUM2
ANSWER [+ X =] 1
END
```

ADD gives ANSWER the form [+ X =] as its first input. The second input represents the number of times the user responds to the question WHAT IS X?. :NUM1, :NUM2, and :RESULT are used by ANSWER's subprocedures ANSWER1 and GETINP. The variables are not given as inputs to ANSWER or its subprocedures. As far as these procedures are concerned, these are global variables. The value of X is still :NUM2.

ANSWER prints the mathematical sentence with the help of ANSWER1. After the sentence is printed, ANSWER asks for the value of X. It then turns the job over to GETINP along with the user's response.

```
TO ANSWER :PHRASE :TIMES
PR []
ANSWER1 :PHRASE
TYPE SE [WHAT IS X?] "\ \
GETINP RL
END
```

```
TO ANSWER1 :PHRASE
PR (SE :NUM1 :PHRASE :RESULT)
END
```

(\ is the way to quote special characters like *space*. ANSWER prints two spaces after the question mark.) GETINP plays an important role. It determines what to do next. If :INP is empty, GETINP assumes this is the user's signal to do something else and so calls MATH1. If :INP is not the same as :NUM2, then GETINP calls ANSWER adding 1 to :TIMES, unless this is the user's third try. On the third try GETINP gives the answer.

```

TO GETINP :INP
IF EMPTY :INP [MATH1 STOP]
IF :NUM2 = FIRST :INP [PR [RIGHT ON] STOP]
IF :TIMES = 3 [PR SE [NOPE, X IS] :NUM2 STOP]
PR [TRY AGAIN]
ANSWER :PHRASE :TIMES + 1
END

```

The MULTIPLY procedure is similar to ADD in structure.

```

TO MULTIPLY :NUM1 :NUM2
MAKE "RESULT :NUM1 * :NUM2
ANSWER [* X =] 1
END

```

A couple of tricks are used here so that ANSWER will work for MULTIPLY, DIVIDE, and SUBTRACT. :NUM2 is always the value of X. :NUM1 is always on the left side of the equals sign and :RESULT is always on the right of the equals sign. What does change is which of these numbers are inputs to a procedure and which are computed in the procedure. For example, SUBTRACT computes the value of NUM1 while :RESULT and :NUM2 are inputs. But the value of X is still :NUM2.

```

TO SUBTRACT :RESULT :NUM2
MAKE "NUM1 :RESULT + :NUM2
ANSWER [- X =] 1
END

```

DIVIDE makes sure that the value of X is always an integer by shifting the role of its first input, which becomes :RESULT. DIVIDE is given :RESULT and computes :NUM1.

```

TO DIVIDE :RESULT :NUM2
MAKE "NUM1 :RESULT * :NUM2
ANSWER [/ X =] 1
END

```

### *Extensions*

There are many modifications you might want to make to this kind of program. The modification I chose is to allow sentences to be in either of two forms.

```

3 + X = 5
X + 3 = 5

```

The changed procedures follow. Notice that the decision as to which form to use is based on whether RANDOM 2 outputs 0 or 1.

```

TO ADD :NUM1 :NUM2
MAKE "RESULT :NUM1 + :NUM2
IF 0 = RANDOM 2 [ANSWER [X +] 0 STOP]
ANSWER [+ X =] 1
END

```

## WORDPLAY

```

TO MULTIPLY :NUM1 :NUM2
MAKE "RESULT :NUM1 * :NUM2
IF 0 = RANDOM 2 [ANSWER [X *] 0 STOP]
ANSWER [* X =] 1
END

```

The new forms for ADD and MULTIPLY are

```

X + 3 = 5
X * 3 = 6

```

where the value of X is still :NUM2.

When SUBTRACT generates a sentence in the form  $6 - X = 2$ , its inputs, :RESULT and :NUM2, are added together to be :NUM1. In the example  $6 - X = 2$ , :NUM1 is 6 and :RESULT is 2.

When the sentence is in the form  $X - 4 = 2$ , then SUB2 computes :NUM2 by adding the inputs :RESULT and :NUM1. In this case :RESULT is 2 and :NUM1 is 4.

```

TO SUBTRACT :RESULT :NUM2
IF 0 = RANDOM 2 [SUB2 :RESULT :NUM2 STOP]
MAKE "NUM1 :RESULT + :NUM2
ANSWER [- X =] 1
END

```

```

TO SUB2 :RESULT :NUM1
MAKE "NUM2 :RESULT + :NUM1
ANSWER [X -] 1
END

```

DIVIDE computes :NUM1 as :RESULT \* :NUM2 when the form is  $6 / X = 2$ . DIVIDE computes :NUM2 as :NUM1 \* :RESULT when the form is  $X / 3 = 2$ .

```

TO DIVIDE :RESULT :NUM2
IF 0 = RANDOM 2 [DIV2 :RESULT :NUM2 STOP]
MAKE "NUM1 :RESULT * :NUM2
ANSWER [/ X =] 1
END

```

```

TO DIV2 :RESULT :NUM1
MAKE "NUM2 :RESULT * :NUM1
ANSWER [X /] 1
END

```

ANSWER needed to be changed as well.

```

TO ANSWER :PHRASE :TIMES
PR []
IF "X = FIRST :PHRASE [ANSWER2 :PHRASE] [ANSWER1 :PHRASE]
TYPE SE [WHAT IS X?] "\ \
GETINP RL
END

TO ANSWER1 :PHRASE
PR (SE :NUM1 :PHRASE :RESULT)
END

TO ANSWER2 :PHRASE
PR (SE :PHRASE :NUM1 "= :RESULT)
END

```



## PROGRAM LISTING

---

```

TO MATH
TS
CT
PR [HERE ARE SOME MATH PROBLEMS.]
MATH1
END

TO MATH1
ADD RANDOM 11 RANDOM 11
SUBTRACT RANDOM 11 RANDOM 11
MULTIPLY 1 + RANDOM 10 1 + RANDOM 10
DIVIDE 1 + RANDOM 10 1 + RANDOM 10
PR [] WAIT 60
MATH1
END

TO ADD :NUM1 :NUM2
MAKE "RESULT :NUM1 + :NUM2
IF 0 = RANDOM 2 [ANSWER [X +] 0 STOP]
ANSWER [+ X =] 1
END

TO MULTIPLY :NUM1 :NUM2
MAKE "RESULT :NUM1 * :NUM2
IF 0 = RANDOM 2 [ANSWER [X *] 0 STOP]
ANSWER [* X =] 1
END

TO SUBTRACT :RESULT :NUM2
IF 0 = RANDOM 2 [SUB2 :RESULT :NUM2 ►
STOP]
MAKE "NUM1 :RESULT + :NUM2
ANSWER [- X =] 1
END

TO SUB2 :RESULT :NUM1
MAKE "NUM2 :RESULT + :NUM1
ANSWER [X -] 1
END

TO DIVIDE :RESULT :NUM2
IF 0 = RANDOM 2 [DIV2 :RESULT :NUM2 ►
STOP]
MAKE "NUM1 :RESULT * :NUM2
ANSWER [/ X =] 1
END

TO DIV2 :RESULT :NUM1
MAKE "NUM2 :RESULT * :NUM1
ANSWER [X /] 1
END

TO ANSWER :PHRASE :TIMES
PR []
IF "X = FIRST :PHRASE [ANSWER2 ►
:PHRASE] [ANSWER1 :PHRASE]
TYPE SE [WHAT IS X?] "\ \
GETINP RL
END

TO ANSWER1 :PHRASE
PR (SE :NUM1 :PHRASE :RESULT)
END

TO ANSWER2 :PHRASE
PR (SE :PHRASE :NUM1 "= :RESULT)
END

TO GETINP*:INP
IF EMPTY :INP [MATH1 STOP]
IF :NUM2 = FIRST :INP [PR [RIGHT ON] ►
STOP]
IF :TIMES = 3 [PR SE [NOPE, X IS] ►
:NUM2 STOP]
PR [TRY AGAIN]
ANSWER :PHRASE :TIMES + 1
END

```

---